Building a Parallel Pipelined External Memory Algorithm Library

Andreas Beckmann*
Institut für Informatik
Goethe-Universität Frankfurt am Main
beckmann@cs.uni-frankfurt.de

Roman Dementiev
Institut für Theoretische Informatik
Universität Karlsruhe (TH)
dementiev@ira.uka.de

Johannes Singler†
Institut für Theoretische Informatik
Universität Karlsruhe (TH)
singler@ira.uka.de

Abstract

Large and fast hard disks for little money have enabled the processing of huge amounts of data on a single machine. For this purpose, the well-established STXXL library provides a framework for external memory algorithms with an easy-to-use interface. However, the clock speed of processors cannot keep up with the increasing bandwidth of parallel disks, making many algorithms actually compute-bound.

To overcome this steadily worsening limitation, we exploit today's multi-core processors with two new approaches. First, we parallelize the internal computation of the encapsulated external memory algorithms by utilizing the MCSTL library. Second, we augment the unique pipelining feature of the STXXL, to enable automatic task parallelization.

We show using synthetic and practical use cases that the combination of both techniques increases performance greatly.

1 Introduction

External memory algorithms process huge amounts of data that do not fit into the main memory of a computer, but must be kept on external memory, usually hard disks. Several models of computation have been proposed for this setting, the most applicable here being the parallel disk model introduced by Vitter and Shriver [14]. Many algorithms have been designed for this model [2, 14, 9], which features the number of I/O operations as its major complexity metric. However, implementing those algorithms is hard, involving a lot of low-level and platform-dependent details. Thus, at least the basic algorithms, which are used as parts of more complex ones, should be provided in a library.

The *Standard Template Library for XXL Data Sets (STXXL)* introduced by Dementiev et al. [8] does just that. It allows easy implementation of algorithms based on the parallel disk model, providing data structures and algorithms, including a priority queue and a sorter. Its interface resembles that of the C++ Standard Template Library (STL) [11]. The STXXL has many virtues that lead to excellent I/O performance, including overlapping of I/O and computation, and abstracted direct access to the disk hardware.

While the clock speed of processors has stagnated recently, the number of disks cheaply attachable to a system and the bandwidth of each disk grows steadily. On modern machines with multiple disks and multiple processor cores, the I/O performance of the STXXL and the I/O efficiency of the algorithms in combination lead to a paradoxical situation. Many external memory algorithms are not I/O-bound, but actually *compute*-bound, e. g. priority queues, breadth-first search, and suffix array construction [6, pp. 48, 111, 163]. This situation is in desperate need for improvement.

In this paper, we describe how to overcome this problem by utilizing multi-core parallelism via the STXXL. Our contribution is twofold. Firstly, we fit out the encapsulated STXXL algorithms and data structures. By using the *Multi-Core Standard Template Library (MCSTL)*, we achieve parallelization of the internal memory work of the algorithms. The same happens for the internal computation of the priority queue, which is a vital *data structure* for many external memory algorithms. Secondly, we extend the pipelining framework of the STXXL. The original purpose of pipelining in the STXXL was to save I/Os by passing between algorithmic components without writing them to disk in-

*Supported in part by MADALGO – Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation and by DFG grant ME 3250/1-1.

†Partially supported by DFG grant SA 933/3-1.

between. However, it also serves very well for modeling external memory algorithms. By using our extensions to the framework, the programmer can automatically exploit the task parallelism inherent in the algorithm.

Outline. We discuss the data parallelization of the STXXL algorithms as a whole, using the MCSTL, in Section 2. The STXXL pipelining and its extension is described in Section 3, also concerning the low-level issues surfacing in this context. We prove the usefulness of our approach in the experimental Section 4, drawing conclusions for future work in Section 5.

1.1 Related work

The basic concept of pipelining is considered folklore in parallel programming. However, using it in this coarse-grained manner is not that common. There exist frameworks to support asynchronous pipelining, e. g. FG [4, 5] and the Intel Threading Building Blocks [10]. FG is targeted to distributed memory systems, though. The pipeline nodes process fixed-length blocks only, which is inflexible. Building non-linear graph structures is not the primary goal of FG, so it is quite inelegant. The Intel Threading Building Blocks also support task parallelism, but are not particularly well-suited for continuously streaming data, i. e. pipelining. The tasks are rather intended to calculate and return a single result.

As a conclusion, none of the frameworks mentioned fully satisfies our requirements.

2 Parallelizing STXXL algorithms using the MCSTL

The MCSTL[1], introduced by Singler et al. [13], is a shared-memory data-parallel implementation of the STL. It parallelizes most of the STL algorithms, featuring load balancing for the embarrassingly parallel routines (like `for_each()`). The combinatorial algorithms include sorting, (multiway) merging, and random shuffling.

Both STXXL and MCSTL provide full genericity through the C++ template mechanism and adhere to the concepts of the STL, like iterators, functors, etc. Since they exploit orthogonal characteristics of the hardware, namely parallel disks and parallel cores, they complement each other ideally.

The STXXL provides several external memory data structures, the priority queue being the one that is compute-intensive. In terms of algorithms, there is comparison-based sorting, integer sorting, random shuffling, and scanning (`for_each()`/`generate()`, and `find()`). Parallelizing these algorithms and data structures was done in two steps:

1. use STL routines instead of hand-written code, where not yet done
2. compile the code with the MCSTL

We describe the details for the comparison-based sort and the priority queue. Sorting is a major step of almost every external memory algorithm, a priority queue is used in many.

For the sorter, the run creation[2] is done using `sort()`, which immediately enables parallel execution when compiled including MCSTL. Choosing the in-place parallel quicksort variant instead of parallel mergesort avoids using additional internal memory.

The run merging step was revised to use `multiway_merge()`, an STL extension of MCSTL, instead of a hand-coded loser tree structure. However, this was not trivial since I/O and computation should be overlapped. Disk access is strictly blockwise and prefetching is obligatory. Naturally, not all input can be kept in memory at the same time. Parallelization, however, requires a partitioning of the data, so all input processed in one call to `multiway_merge()` must be available from the beginning, and no sequence may run empty. This leads to the following sub algorithm, which is executed as long as there are elements left.

1: **while** not all sequences empty **do**
2: find the minimum element m over all sequence maxima (regarding only the respective current block), let s be the sequence m is contained in
3: rank m in all sequences except s using binary search and calculate the number of elements n which can be processed before s runs empty
4: multiway-merge n elements
5: swap in the block that has been fetched as next block of s in the meanwhile and start prefetching the next block for s
6: **end while**

In a similar way, parallel versions of `sort()` and `multiway_merge()` were integrated into the priority queue implementation, which is based on sequence heaps [12].

The part of our work described in this Section 2 is referred to as *STXXL Parallel Algorithms*. Its implementation has added only less than 10% in lines of code to the STXXL.

[1]Development of the MCSTL is ongoing as a part of the GNU C++ compiler (as of version 4.3), namely the libstdc++ parallel mode. We used MCSTL version 0.8.0-beta here.

[2]Efficient external sorting usually consists of two major steps: run creation and run merging. First, the data is sorted internally and written back to disk, run after run (one run filling the whole internal memory). Second, the runs are merged from disk to disk.

3 Task parallelism through asynchronous pipelining

3.1 STXXL traditional pipelining

The STXXL supports passing data between algorithmic components without writing them back to disk in-between. This technique is referred to as *pipelining*. Pipelining is programmed by setting up a directed acyclic *flow graph*.

Nodes of the flow graph represent components that process data. The edges denote the flow of data, indicating its direction. Its structure is usually fixed in the source code, the type dependencies being resolved at *compile time*.

A node may either be a *scanning* node (drawn as square), a *sorting* node (drawn as funnel), or a *file* node (drawn as circle). They process *elements* of possibly different type and number. Scanning operations consume one element after the other, and emit zero or more elements (typically one per element read), but do all work in-order. Their actual functionality is defined by the programmer, e. g. applying some kind of reduction on the elements, filtering elements out, or changing the elements themselves. The sorting nodes sort the elements with respect to some total order, storing them to disk temporarily. File nodes either read elements from disk and provide them to their respective successor, or consume elements from their predecessor and store them onto disk.

With these three kinds of nodes, almost all efficient external memory algorithms can be represented. This is because random access is very inefficient due to disk latency, it has to be avoided at all cost. Sorting many elements, however, is reasonably efficient. Scanning many elements in-order is the most efficient operation, of course.

In the flow graph, elements can be *pulled* (drawn as solid arrows) or *pushed* (drawn as dashed arrows). For that reason, we have two interfaces for nodes. When its `operator*()` is called, a node implementing the *pull interface* (shortly called *pull node*) reads data from its predecessor(s), processes it, and returns the result to the caller. Checking whether the next element exists is done via `empty()`, advancing via `operator++()`. This is similar to the input iterator concept of the STL. Symmetrically, an element is passed to a node implementing the *push interface* (shortly called *push node*) via `push()`. It is then processed, and the result is usually pushed to its successor(s). A counterpart to `empty()` is not necessary, since a push node always has to accept more data. Conversely, we have to provide a call to notify the node of the end of data, namely `push_stop()`.

Branching a data stream can be achieved by a node pushing into additional successor nodes while being pulled. Joining is achieved by making a node pull multiple predecessors, before returning the (somehow combined) result.

A flow graph needs to have a *primary sink*, usually a file node. Execution of the pipeline is triggered by starting to fill the primary sink using the `materialize()` function call. In order to get data, the primary sink then pulls data from its predecessor in the data flow graph. The predecessor of the primary sink recursively pulls from its predecessor(s), and pushes to other successors, putting the whole pipeline into action, up to the source node(s). To ensure that all parts of the flow graph are active, all nodes must be reachable from the primary sink by traversing pull edges backwards and push edges forwards. This implies (each node can have at most one pulling out edge) that there is exactly one primary sink in a valid flow graph.

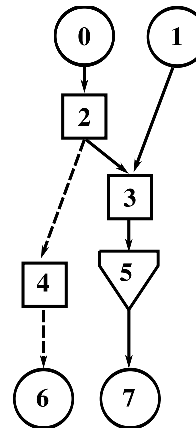


Figure 1: Basic example for pipelining.

We subsume all the introduced terminology using the example from Figure 1. Nodes 0 and 1 are (source) file nodes, while nodes 6 and 7 are (sink) file nodes. Node 7 is the primary sink, while node 6 is not, since not all nodes can be reached starting from node 6 (in fact, no node at all can be reached). Nodes 2, 3 and 4 are scanning nodes. When node 2 is pulled by node 3, it pushes side-ways to node 4, which in turn pushes to file node 6. Node 3 pulls from both node 2 and node 1, combining the received data into a sequence that is sorted subsequently by node 5.

The glue code for setting up this example pipeline is given in Appendix A.

Sorting nodes (actually their run creation part) also form sinks, implying a `materialize()` step at node construction time. Therefore, in the traditional STXXL pipelining, the sorting nodes had to be split into a (pushed) run creator node and a (pulled) run merger node if there were multiple sorters fed from a common source. This construct is called a *split-up sorting node*, see Figure 4 for an illustration. Otherwise, the first sorting node constructed would have started pulling data before the other nodes were even constructed. In fact, this is already impossible to compile because it leads to a circular type definition.

3.2 STXXL parallel pipelining

Many algorithmic problems [6] have been solved successfully using the pipelining approach. Therefore, it is promising to speed up things on a conceptual level, as described in this paper.

There exist two opportunities for task parallelism in the context of pipelining. For a more intuitive understanding, assume that the nodes are ordered topologically from top to bottom, the source(s) at the very top, the sink(s) at the very bottom, data always streaming downwards.

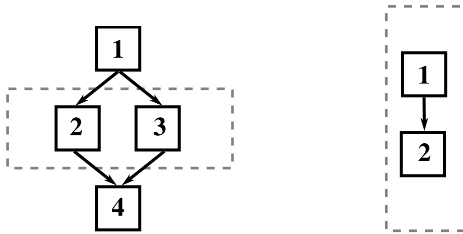


Figure 2: Horizontal (left) and vertical (right) task parallelism for nodes 1 and 2.

Horizontal task parallelism: Nodes that are not connected (respecting the normal direction of the edges) do not have any data dependency. Thus, they can be executed in parallel.

Vertical task parallelism: Let scanning node N_1 be an (indirect) predecessor of scanning node N_2 with respect to the flow graph (respecting the normal direction of the edges). Then, there exists a data dependency between the two nodes. However, since the elements are accessed in order, both computations can be overlapped and thereby parallelized. Each node fetches a part of the input from its predecessor(s), processes it, and passes it on to its successor.

Figure 2 show the two situations described.

We now supplement the STXXL pipelining framework with so-called *asynchronous* nodes, which can be used by the programmer to decouple two algorithmic nodes by inserting an aynchronous node in-between. The asynchronous nodes process data by spawning a worker thread, communicating with their predecessors and successors to control the data flow. This automatically enables parallel execution in both cases stated above.

The asynchronous nodes come in three flavors:

- An *async pull node* pulls data from its predecessor(s) and processes it asynchronously in a worker thread, storing the resulting elements for availability to its successor.
- Symmetrically, an *async push node* gets elements pushed into. It processes and stores the resulting elements. Asynchronously, resulting elements are pushed into the successor.
- A *push-pull node*, finally, does not do work actively, but just synchronizes two nodes, one pushing into it, the other pulling from it.

To make this work sharing efficient, each asynchronous node needs two element buffers, an producer buffer and a consumer buffer. The producer buffer absorbs resulting elements (or just input elements for the push-pull node), while the consumer buffer is used for serving the successor node. Routines may block when there is no data readily available. When the producer buffer is full and the consumer buffer is empty, the two buffers are swapped, and operation can continue (see Figure 3). Waiting for these two events is done using OS-supported synchronization, so no CPU power is wasted. The buffer is needed to amortize the overhead for synchronization of the threads. Synchronizing for every single element would be far too time-consuming. Finally, the buffer size is a tuning parameter, blending these two extremes one element and all elements. Larger buffers amortize the synchronization overhead better, but decrease parallelism because of the implied latency.

Unfortunately, the insertion of asynchronous nodes cannot happen without changes to the source code of an already implemented algorithm. This is because of the very tight coupling of nodes at compile time, which is necessary for good performance[3]. Routines like `operator*()` immediately call the same routine of the predecessors, so it is impossible to insert asynchronous nodes automatically.

In addition to the asynchronous pipeline nodes, we also integrated overlapping of reading the input and sorting the runs for the pipelined sorter, which was a shortcoming of the existing version. This effectively adds an async pull node to each sorter, located right in front of the run creator (see Figure 5). Differently to a usual async pull node, this one's buffer is necessarily as large as a whole run size, since the run creation needs a complete run of data before it can start. By default, the merging is not done asynchronously, because this relatively cheap task combines well with subsequent scanning nodes. If the programmer desires otherwise, he can just add an async pull node right after the sorting node.

For an usage example see the *Diamond flow graph* in Figure 6, disregarding the asynchronous nodes (drawn as gear wheels) at first. This flow graph is a example for an external memory algorithm that could well exploit task parallelism. File node 0 contains a sequence of pairs. In

[3]The compiler is able to fold several stages into one call using inlining, which is crucial for performance.

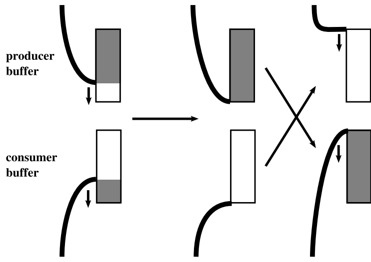


Figure 3: Swapping buffers.

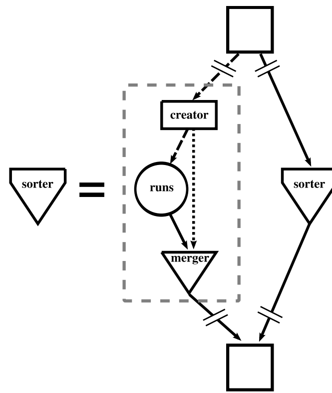


Figure 4: Split-up sorting node in context.

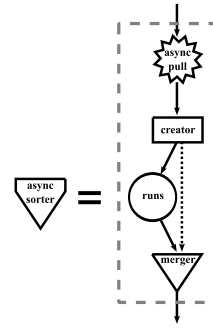


Figure 5: Equivalent for asynchronous sorting node.

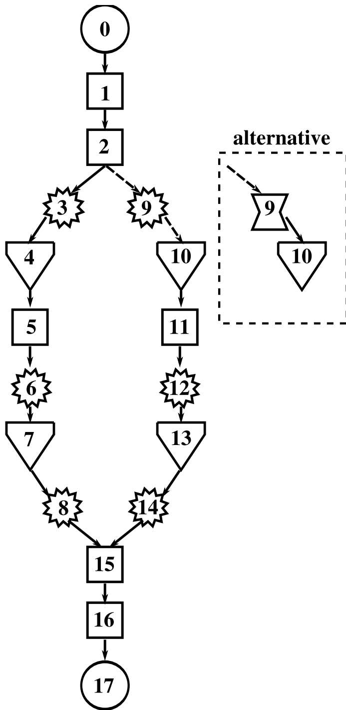


Figure 6: Diamond flow graph.

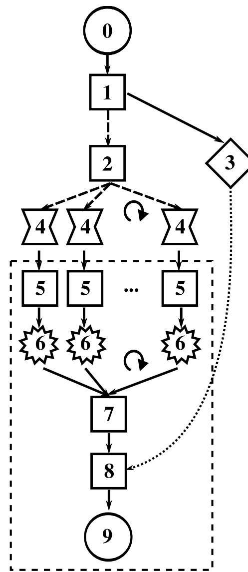


Figure 7: Distribute-Collect flow graph.

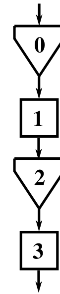


Figure 8: doubling/ quadrupling flow graph.

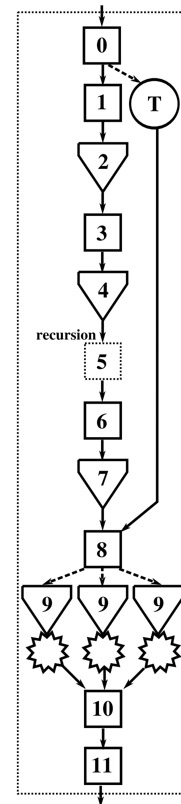
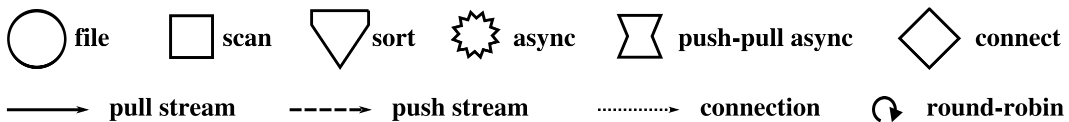


Figure 9: DC3 flow graph.



node 2, the stream is doubled, after calculating a checksum in node 1. Nodes 4 and 10 sort by the second value of the pair, ascending and descending, respectively. Analogous sorting by the first value of the pair is done in nodes 7 and 13, after taking the checksums in nodes 5 and 11, again. Node 15 rejoins the data into pairs of pairs, a checksum is again taken in node 16. The Diamond flow graph is quite typical for a data flow graph since the sorting nodes, consuming most of the computational effort, are connected with each other by relatively cheap scanning nodes.

The traditional pipelining executes the paths between nodes 4 and 7 and between the nodes 10 and 13 *non-concurrently*, one after the other. To enable task-parallel execution of the program, the async pull nodes 3, 6, 8, 12, and 14 and the async push node 9 are introduced. In fact, the nodes 3, 6, 9, and 12 are integrated into their successive sorting nodes, by using async sorting nodes. Adding the others is very easily programmed, adding two lines of code per asynchronous node, and changing two more.

Following from that, the following sets of nodes will run in parallel, in consecutive phases: *1st phase:* 0, 1, 2, 3, run creation part of 4, 9, run creation part of 10; *2nd phase:* run merging part of 4, 5, 6, run creation part of 7, run merging part of 10, 11, 12, run creation part of 13; *3rd phase:* run merging part of 7, run merging part of 13, 15, 16, 17. The number of phases is defined by the maximum number of sorting nodes on a path from a source to a sink (two in this case), plus one. All inherent task parallelism is automatically exploited.

More elegant flow graph design. As mentioned before, the traditional STXXL pipelining starts running a node already at its construction time. However, this can happen at a point in time where the flow graph is still incomplete. This is undesirable, since it inhibits horizontal parallelism, forces split-up sorting nodes, and would also forbid analysis of the flow graph as a whole. Therefore, we introduce a `start()` call, which triggers processing, i. e. thread creation for the asynchronous nodes. It can be called after construction the whole flow graph.

The sorting nodes can be used symmetrically now, all being of the same generic type. Starting the nodes is done in a depth-first manner after the flow graph has been fully constructed, again starting at the primary sink.

This raises the question, whether in the course of adding parallel pipelining, we can completely abandon to *push* elements. Multiple consumers nodes could *pull* from an emitting node, thereby implicitly splitting the stream. However, this is ineffective for several reasons. First of all, the data emitted to different successors could be different in type, size, and/or volume. This would force the respective node to emit tuple types being the union of all requested outputs, each successor selecting its respective component. Hence, the required internal memory bandwidth would be unnecessarily high. Secondly, advancing the emitting node to the next element would have to be synchronized among all consuming nodes. Doing this for every single element imposes a huge system overhead, which renders asynchrony useless. Even synchronizing for a large batch of elements, as described in Section 3.3, would not help without interconnecting a buffer. As a consequence, we have to keep pushing elements, at least into a push-pull node.

For the Diamond flow graph, a push-pull node (drawn as waisted square) could be inserted at position 9, while changing node 10 from having a pushed run creator to a usual sorting node. There are only negligible differences in execution between both alternatives. However, concerning programming, while the first one stays backward-compatible, i. e. the asynchronous nodes can be taken out without losing liveness, the second alternative is more symmetric and thus more elegant.

Round-robin task parallelism. Generating random input for the tests arose to be a bottleneck. Computations can happen completely independently, the results feeding further nodes in arbitrary order, which can also happen in other circumstances (e. g. processing input from independent internal-memory sources, where intermixing of the produced elements does not harm). To enable parallelization in such a case, we added *distribute* and *collect* node types. While the former pushes elements to several nodes of the same type in a round-robin fashion, the latter collects them in the very same way, keeping the overall order. Executing each of the intermediate nodes by a separate thread parallelizes the computation.

To make every node reachable from the primary sink with the edge directions stated above, a special *connect node* has to be inserted, see Figure 7 for an example (connect node 3 between nodes 1 and 8). The connect node has an additional edge outgoing to another node (drawn as dotted line). This node starts the connect node before its actual predecessor(s). However, no data is transferred across the edge ever. Instead of the distribute-collect construct, we also could have used an input node with a parallelized `generate()` call, but this would request a natively parallel random number generator.

The work described in this Section 3.2 is referred to as *STXXL Asynchronous Pipelining*.

3.3 Implementation details

Threading support. While the MCSTL nicely benefits from OpenMP for its data-parallel routines, fork-join parallelism is not sufficient for the pipelining's task parallelism. Here, we use the usual OS threading (POSIX threads in this case).

Low-level issues concerning the pipelining layer. The pipelining approach can lead to constant-factor computation overhead due to high-level programming, i. e. passing only one element at a time from one node to the next. For small types, this streaming bandwidth can be even lower than the I/O bandwidth, which is unacceptable here because it contradicts speeding up internal computation. Therefore, we had to think about modifying the existing code in order to provide small-overhead streaming, while keeping the interface mostly backward-compatible.

As mentioned above, the traditional STXXL pipeline interface (for pulling) consists of the three methods `empty()`, `operator*()`, and `operator++()`, resembling an input iterator interface. We extended this interface by a batch-wise access method. The functions `batch_length()`, `batch_begin()`/`operator[]()` and `operator+=()` do the same things as their counterparts, but for a chunk of elements at a time. `batch_begin()` returns a random-access iterator (a typed pointer in the best case), that allows efficient processing of many elements without calling back the predecessor node explicitly, in particular without calling `empty()` every single time. For backward compatibility to nodes that support only emitting one element at a time, the programmer just inserts an asynchronous node which pulls individual elements, but allows batch access to the successor node. Symmetric methods exist for push nodes.

The asynchronous nodes support the batch methods particularly well using arrays as buffers.

4 Experiments

To evaluate the practical performance of our implementation, we conducted synthetic tests to measure the principal performance, as well as an algorithmic application of the library, to show the effect in a use-case.

Platform. We tested our programs on an Intel Xeon E5345 machine (2 sockets, 2 × 4 cores at 2.33 GHz, Intel Core 2 micro-architecture) with 8 hard disks (about 72 MB/s I/O each).

At the time, GCC did not yet support nesting OpenMP parallelism in multiple OS threads[4]. Therefore, we used the Intel C++ Compiler 10.0.026 and 10.1.011 with optimizations switched on (-O3), accessing the GCC 4.2 STL implementation.

Synthetic tests. We tested comparison-based sorting and priority queues, as well as the Diamond flow graph, comparing the traditional version of STXXL against the incrementally improved ones. We took 100 GB of input data (pairs of 64-bit unsigned integers), and 1 GB of internal memory for the run creation and run merge nodes.

[4]This will be supported as of version 4.4.

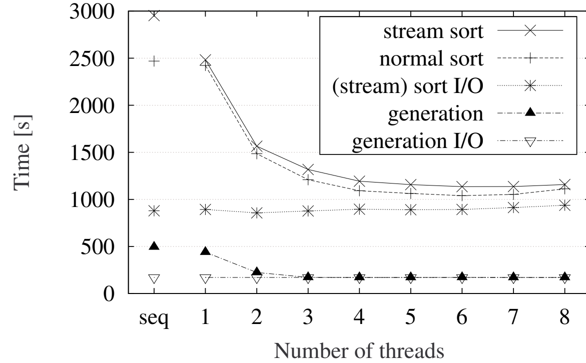


Figure 10: Sorting 100 GB of 64-bit unsigned integer pairs. The I/O lines shows for how much time the I/O was active during execution.

Plain Sorting. As shown in Figure 10, the improvements in running time for sorting are decent. While a single thread takes 2 423 s to sort the data, the 4-thread version needs only 1 092 s. There is no further speedup for more threads, since the parallel sorting is in fact memory-bandwidth-bound from that point on, for this data type. Still, we get pretty close to the I/O-limited minimum running time of about 880 s, almost fully loading the machine. The speedups for smaller or harder to compare elements would be even better, since the bandwidths are the limiting factors.

For the STXXL parallel pipelining variant, the differences between a monolithic sort call, and a sort call embedded in a pipelined flow graph, are now reduced to few percents. The traditional pipelined sorter, which does not support overlapping of reading and sorting in the run creation step, clearly performs worse (2 953 s).

Priority Queue. Priority queues are the essential part of external memory algorithms that use *time forward processing* [3, 1], taking most of the processing time in these applications. Time forward processing means using a priority queue to gather data elements that are required for a future processing step, by using an step identifier as key.

Our test case first increases the number of contained elements by calling (`insert()`, `delete-min()`, `insert()`) in a loop, until the priority queue size of 100 GB is reached. Then, the queue is emptied again by calling (`delete-min()`, `insert()`, `delete-min()`), symmetrically. The inserted elements of 8 byte size are chosen to be a random amount larger than the element deleted last, i. e. we have a *monotone* priority

queue, which is always the case for time forward processing. We set an internal memory limit of 4 GB.

Threads	1	2	4	8
Filling Phase	9 031	6 615	5 564	5 099
Removal Phase	5 260	3 921	3 291	3 068
Total (speedup)	14 291	10 536 (1.36)	8 855 (1.61)	8 167 (1.75)

Table 1: Processing a priority queue with 8 byte elements, of maximum size 100 GB. Timings are in seconds.

The result are stated in Table 1. Using only 2 threads, the total running time decreases from 14 291 s to 10 536 s, being equivalent to a 36% relative speedup. The speedup increases to 61% and 75% for 4 and 8 threads, respectively. This is quite impressive for a data structure handling such fine-grained requests.

Generating random input by using distribute/collect. We consider the generation of random input data first. For each thread, there is an asynchronous node, pulling random numbers from another node. Those are collected in a buffer-wise round-robin fashion, similar as shown in Figure 7 (only the part inside the dashed rectangle is needed here). The running time decreases from 494 s of the traditional pipelining implementation (TP), to 171 s with 3 or more threads, being strictly I/O-bound from that point on.

Threads	1	3
TP (speedup)	11 124	6 098 (1.82)
BP (speedup)	10 944 (1.02)	5 965 (1.86)
PP (speedup)	4 629 (2.40)	3 057 (3.64)

Table 3: Executing the Diamond flow graph with different STXXL and machine configurations. Input data size 100 GB. Timings are in seconds.

Diamond flow graph. To evaluate how effectively data parallelism and task parallelism can join forces, we have run tests with the conceptual pipelined algorithm represented by the Diamond flow graph in Figure 6.

Executing the Diamond flow graph with TP, processing 100 GB of data takes three hours on our machine (see Table 3). If we allow 3 threads for each sorting node (run creator) using the parallelized sort algorithm, the running time almost halves. Adding the batching (BP) improves the performance a bit more, for both the sequential and the parallel case. However, when we add all the asynchronous nodes to gain full parallel pipelining (PP), many nodes can work simultaneously, dropping the running time to 4 629 s for one thread per sorting node, and finally to less than one hour (3 057 s) with three threads per sorting node. This corresponds to a speedup of more than 3.6 overall. The maximum number of *concurrent* threads is actually 8, two teams of 3 sorting the current runs, two more filling the upcoming runs. Since the fastest possible I/O time is about 2 500 s already, most of the potential speedup is attained.

4.1 Application test

We also evaluated our work with an algorithmic application, namely external memory suffix array construction. Suffix array construction is an important sub step of many string processing algorithms, e. g. Burrows-Wheeler transform.

Several variants had already been implemented using STXXL pipelining [7]. We inserted asynchronous nodes and enabled the STXXL Parallel Algorithms.

The input data consists of prefixes of the “Source” and the “Gutenberg” data set, as in [7]. As a reference, the traditional pipelining (TP) of the STXXL is used. We added STXXL Parallel Algorithms (P), sorters with asynchronous reading (A), and their combination (A+P). Finally, we manually inserted additional asynchronous nodes into the algorithm and tested with and without STXXL Parallel Algorithms (A+M/A+M+P). The numbers of threads were chosen such that all cores were loaded. The results are stated in Table 2.

For the *doubling* algorithm, speedups up to 2.51 could be achieved by combining STXXL Parallel Algorithms and sorters with asynchronous reading. No additional improvements are gained by manually adding asynchronous nodes because of the simple structure (see Figure 8) of the algorithm. One iteration of the loop consists of a single straight forward path of a runs merger, full sorter and runs creator with two scanning nodes in-between. Therefore the asynchronous nodes in the run formation part of the sorters already gain a maximum of task parallelism. The *quadrupling* algorithm is a variant of the doubling algorithm that processes larger elements in the same data flow graph (Figure 8). The scanning and sorting nodes require more internal work per element, but the number of iterations and I/Os needed to compute the suffix array is reduced, resulting in a faster algorithm [7]. The speedup achieved by using STXXL Parallel Algorithms and sorters with asynchronous reading is 1.88. Since no improvements were expected, manually adding asynchronous nodes has not been tested on the quadrupling algorithm.

input data	Source								Gutenberg							
MEM	1 024 MB								2 048 MB							
algorithm	doubling				DC3				quadrupling				DC3			
input length	2^{28}		2^{29}		2^{28}		2^{29}		2^{29}		2^{30}		2^{29}		2^{30}	
	[s]	sp.	[s]	sp.	[s]	sp.	[s]	sp.	[s]	sp.	[s]	sp.	[s]	sp.	[s]	sp.
TP	2 462	1	5 087	1	864	1	1 748	1	3 587	1	7 546	1	1 820	1	3 664	1
P	1 097	2.24	2 357	2.16	589	1.47	1 217	1.44	1 981	1.81	4 333	1.74	1 227	1.48	2 525	1.45
A	1 946	1.26	3 775	1.35	776	1.11	1 505	1.16	2 680	1.34	5 206	1.45	1 646	1.11	3 171	1.16
A+M	1 924	1.28	3 751	1.36	676	1.28	1 277	1.37	–	–	–	–	1 485	1.23	2 774	1.32
A+P	999	2.46	2 023	2.51	572	1.51	1 144	1.53	1 911	1.88	4 266	1.77	1 217	1.50	2 367	1.55
A+M+P	998	2.47	2 036	2.50	558	1.55	1 123	1.56	–	–	–	–	1 201	1.52	2 249	1.63

Table 2: Suffix array creation using different algorithm and library combinations. MEM gives the maximum amount of internal memory the algorithms were allowed to use for sorting and pipeline buffering, *sp.* stands for *speedup*. This amount is shared by all concurrent sorters and asynchronous pipelines.

In the *difference cover* (DC3) algorithm (see Figure 9), the data flows through three parallel paths between nodes 8 and 10, which contain sorters and benefit from running asynchronously. Additional asynchronous nodes are inserted after these sorters (node 9), running the merging steps in parallel, too, before the paths are joined in node 10. The pipeline shown here is actually instantiated recursively, reducing the number of elements by one third each time. The total speedup of DC3 is limited to 1.63 by the sequential scanning nodes, requiring more explicit parallelization. Overall, the DC3 algorithm is still the fastest, and also sped up significantly.

We also note that the combined strategies achieve better speedups for bigger inputs. In detail, the results show that each of the improvements to the STXXL is worthwhile, as it improves performance.

5 Conclusion and future work

In this paper, we have shown how to catch up the margin that disks have gained over single-core processing power, by combining two libraries and adding task-parallel pipelining. STXXL Parallel Algorithms and STXXL Asynchronous Pipelining combines to *STXXL Parallel Pipelining*, enabling both data parallelism and task parallelism automatically. To our knowledge, there is no other library providing integrated support for parallel external computation. Our experiments, both synthetic and applied, have proven the potential of our approach, achieving considerable speedup in both synthetic and application tests. This is despite the fact that only little had to be changed in the source codes. Library users can benefit from the improvements very easily, even for existing code.

5.1 Future work

The memory assigned to each node (both working and asynchronous nodes) must still be manually configured by the programmer. This could be done automatically with respect to a global memory limit. The nodes should be queried by a to-be-defined interface for their memory requirements and desires. Then, the available memory could be split in a fair way, using some optimization strategy. For this purpose, the flow graph would have to be segmented into the phases that have to run consecutively. The optimizer could also decide to run less nodes than possible in a phase, in order to have more memory available for critical nodes (e. g. in order to save a recursive merge pass for sorting).

Similarly, the allotments of threads to nodes could be handled more intelligently. So far, all parallel algorithm nodes are executed with a fixed specified number of threads. Again, by analyzing the graph and requesting information from the nodes about the effectiveness of more parallelism, the available processor cores could be used with best effect.

Although we discussed directed *acyclic* pipelining graphs here, with asynchronous nodes, having cyclic graphs is not off-limits any longer. A loop could be driven by an asynchronous node, without the need for a sink. Termination would be induced by a node emitting no further elements.

We are currently investigating distributed-memory parallel operations in conjunction with STXXL.

References

[1] L. Arge. The Buffer Tree: A New Technique for Optimal I/O-Algorithms. In *4th Workshop on Algorithms and Data Structures*, number 955 in LNCS, pages 334–345, 1995.

[2] L. Arge, G. S. Brodal, and L. Toma. On external-memory MST, SSSP and multi-way planar graph separation. *Journal of Algorithms*, 53, 2004.

[3] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamasia, D. E. Vengroff, and J. S. Vitter. External memory graph algorithms. In *6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, 1995.

[4] T. H. Cormen and E. R. Davidson. FG: A framework generator for hiding latency in parallel programs running on clusters. In *Proceedings of the ISCA 17th International Conference on Parallel and Distributed Computing Systems (ISCA PDCS)*, pages 137–144, 2004.

[5] E. R. Davidson and T. H. Cormen. Building on a framework: Using FG for more flexibility and improved performance in parallel programs. In *19th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2005.

[6] R. Dementiev. *Algorithm Engineering for Large Data Sets*. PhD thesis, Universität des Saarlandes, 2006.

[7] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. In *Proceedings of the Seventh Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 86–97, 2005.

[8] R. Dementiev, L. Kettner, and P. Sanders. STXXL: standard template library for XXL data sets. *Software: Practice and Experience*, 38(6):589–637, 2008.

[9] R. Dementiev and P. Sanders. Asynchronous parallel disk sorting. In *15th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 138–148, 2003.

[10] Intel Threading Building Blocks website. `http://osstbb.intel.com/`.

[11] P. J. Plauger, A. A. Stepanov, M. Lee, and D. R. Musser. *The C++ Standard Template Library*. Prentice-Hall, 2000.

[12] P. Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5, 2000.

[13] J. Singler, P. Sanders, and F. Putze. The Multi-Core Standard Template Library. In *Euro-Par: Parallel Processing*, volume 4641 of *LNCS*, pages 682–694, 2007.

[14] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I/II. *Algorithmica*, 12(2/3):110–169, 1994.

A Source code examples

Source code for a simple linear pipeline.

```
// input from external vector, implicit source file node
typedef typeof(streamify(input.begin(), input.end())) input_node_type;
input_node_type input_node = streamify(input.begin(), input.end());

// compute checksum over all elements: scanning node
typedef checksummer<input_node_type> checksum_node_type;
checksum_node_type checksum_node(input_node);

// double the value of all elements: scanning node
typedef doubler<checksum_node_type> doubler_node_type;
doubler_node_type doubler_node(checksum_node);

// connection to sorting node
typedef sort<doubler_node_type, comparator_type> sort_node_type;
sort_node_type sort_node(doubler_node, comparator, run_size);

// output to external vector, implicit sink file node
materialize(sort_node, output.begin(), output.end());
```

Same pipeline as above, with an async pull node added between the two scanning nodes. The changes are highlighted.

```
// input from external vector, implicit source file node
typedef typeof(streamify(input.begin(), input.end())) input_node_type;
input_node_type input_node = streamify(input.begin(), input.end());

// compute checksum over all elements: scanning node
typedef checksummer<input_node_type> checksum_node_type;
checksum_node_type checksum_node(input_node);

// introduced async pull node
typedef pull<checksum_node_type> pull_node_type;
pull_node_type pull_node(buffer_size, checksum_node);

// double the value of all elements: scanning node
typedef doubler<pull_node_type> doubler_node_type;
doubler_node_type doubler_node(pull_node);

// connection to sorting node
typedef sort<doubler_node_type, comparator_type> sort_node_type;
sort_node_type sort_node(doubler_node, comparator, run_size);

// output to external vector, implicit sink file node
materialize(sort_node, output.begin(), output.end());
```